AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions

Qiangyu Pei[†]
Huazhong University of Science and Technology
Wuhan, China
peiqiangyu@hust.edu.cn

Yongjie Yuan[†]
Huazhong University of Science and Technology
Wuhan, China
jayayuan@hust.edu.cn

Haichuan Hu
Huazhong University of Science and Technology
Wuhan, China
huhc@hust.edu.cn

Qiong Chen
Huawei
Hangzhou, China
chenqiong13@huawei.com

Fangming Liu[*]
Peng Cheng Laboratory
Huazhong University of Science and Technology
Wuhan, China
fangminghk@gmail.com

ABSTRACT

Recent advances in deep learning (DL) have spawned various intelligent cloud services with well-trained DL models. Nevertheless, it is nontrivial to maintain the desired end-to-end latency under bursty workloads, raising critical challenges on high-performance while resource-efficient inference services. To handle burstiness, some inference services have migrated to the serverless paradigm for its rapid elasticity. However, they neglect the impact of the time-consuming and resource-hungry model-loading process when scaling out function instances, leading to considerable resource inefficiency for maintaining high performance under burstiness.

To address the issue, we open up the black box of DL models and find an interesting phenomenon that the sensitivity of each layer to the computing resources is mostly anti-correlated with its memory resource usage. Motivated by this, we propose *asymmetric functions*, where the original Body Function still loads a complete model to meet stable demands, while the proposed lightweight Shadow Function only loads a portion of resource-sensitive layers to deal with sudden demands effortlessly. By parallelizing computations on resource-sensitive layers, the surging demand can be well satisfied, though the rest of the layers are performed serially in Body Functions only. We implement asymmetric functions on top of Knative and build a high-performance and resource-efficient inference serving system named AsyFunc with a new auto-scaling and scheduling engine. Evaluation results driven by production traces show that compared with the state of the art, AsyFunc saves computing and memory resources by up to 31.1% and 32.5%, respectively, while providing consistent performance guarantees under burstiness.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

serverless, inference system, workload burstiness, resource-management

ACM Reference Format:
Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620678.3624664

[†]Both authors contributed equally to this research.

[*]Corresponding author: Fangming Liu (fangminghk@gmail.com). Q. Pei, Y. Yuan, and H. Hu are with the National Engineering Research Center for Big Data Technology and System, the Services Computing Technology and System Lab, Cluster and Grid Computing Lab in the School of Computer Science and Technology, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan, China. Q. Chen is with the YuanRong Team of Distributed Lab at the 2012 Laboratory Central Software Institute, Huawei, Hangzhou, China. F. Liu is with Peng Cheng Laboratory, and Huazhong University of Science and Technology, China.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624664

1 INTRODUCTION

Deep Learning (DL) has enabled a variety of intelligent applications in recent years, from virtual assistants to smart traffic management. According to data statistics from China's largest local life service platform, millions of queries are processed every minute using DL models [60]. On Facebook alone, more than 200 trillion inference queries are processed daily [32]. Typically, these models go through two phases for each application scenario: offline *training* to achieve the desired accuracy by iteratively tuning the model parameters, and online *inference* to perform user-facing tasks in real-time. In contrast to training, model inference imposes strict requirements on real-time performance, especially the end-to-end latency specified by service level objectives (SLOs) [10, 62]. For example, 98% of user requests should complete within 200 ms, and SLO violations result in a poor user experience and potentially lower business revenue [62]. A recent report shows that a 100 ms increase in latency can lead to a 1% decrease in revenue [23].

However, the ubiquitous *bursts* of user requests make it difficult to maintain desired SLOs [2, 40] as more resources are suddenly requested but are not available at the moment (e.g., occupied by another service or idle but not initialized). A common practice to deal with bursts is *over-provisioning*, i.e., preparing sufficient resources in advance, which would result in a considerable waste of resources during valley periods [59]. The problem posed by bursts is exacerbated in an inference platform because DL models are generally resource-hungry. For example, the GPT-3 model [8] consumes 325 GB of memory for storing its parameters, as well as necessary computing resources to run the inference, meaning that prohibitive amounts of resources need to be allocated in advance. This overhead increases dramatically as more DL models are served simultaneously [25] and as the models grow larger over time, especially with the recent emergence of large language models such as ChatGPT [36]. Therefore, a question emerges about *providing high-performance yet resource-efficient inference services despite bursts*.

There have already been some attempts at this question. Amazon SageMaker [5] is a well-known inference platform that uses virtual machines (VMs) to execute inferences. Despite low operational costs, the bulky VMs make scaling too slow to meet real-time requirements during sudden spikes in requests. More recently, *serverless computing* offers opportunities for dealing with bursts [57]. For example, MArk [62] combines sparse VMs and elastic serverless functions[1] for model serving under bursts, demonstrating the potential benefits of serverless. Industry products such as Amazon Alexa [4] and Netflix content delivery [44], have also gradually deployed their services on serverless platforms, which can respond to fluctuating workload levels in a quick and cost-effective manner due to the rapid elasticity and fine-grained billing that serverless offers [2, 25, 60].

Despite the prominent advantage of serverless in handling bursts, we note that the unavoidable model-loading process when creating new function instances significantly limits the benefits of auto-scaling for the following two reasons. (1) *High model-loading latency invalidates the reactive on-demand scaling policy.* According to our measurements, the model-loading latency can be 2 to 50 times the inference latency, making real-time services impossible during bursts. (2) *High resource requirements prevent the proactive prediction-based scaling policy.* Since a DL model can consume hundreds to thousands of megabytes of memory, prewarming a sufficient number of function instances in advance can result in significant resource consumption. Given the unpredictability of future requests, the pre-warmed instances typically far surpass the actual demands [45]. These issues ultimately preclude high performance while resource-efficient inference services under bursts.

By solving the scaling issue as a consequence of the time-consuming and resource-hungry model-loading process, we aim to arm serverless functions with resource-efficient scaling capabilities while maintaining consistent performance. Rather than viewing the entire DL model as a complete black box, as has been the case in previous works [2, 25, 60], we identify unique opportunities that arise from the heterogeneous behavior of the internal layers. In particular, we find that the sensitivity of each layer to computing resources is almost negatively correlated with its parameter size, as shown later in Figure 2. This implies that loading a small number of resource-sensitive layers can achieve comparable inference latency as loading a complete model, but significantly reduces the overhead of loading the model when provisioning a new instance. For example, for one of the latest object detection models YOLOv8x [43], if the top 10% most resource-sensitive layers get loaded, the model-loading time and memory consumption can be reduced from 163 ms and 261 MB to 13.6 ms and 17.8 MB, respectively, while the inference latency can remain almost unchanged by allocating a few more CPU cores temporarily.

Driven by the observation on the completeness of DL models, we propose a fine-grained layer-level scaling policy in combination with the existing coarse-grained model-level scaling policy. The former scales out functions with a portion of resource-sensitive layers which we call *Shadow Function*, and the latter scales out functions still with a complete model which we call *Body Function*. Body Function basically loads a complete model to maintain consistent performance under

[1]The serverless function contains a piece of user code and can be instantiated as an individual execution unit.

stable demands, and adjusts periodically to long-term fluctuations in workload levels (e.g., 1 minute). In comparison, Shadow Function loads only a portion of resource-sensitive layers so that it can respond quickly to sudden demands. When a burst arises, it will be too slow to provision a new Body Function, but the Shadow Function can be provisioned in a timely manner. By pairing an existing Body Function with a new "asymmetric" Shadow Function, they can perform inference executions on the resource-sensitive layers together, while only the Body Function is still responsible for the rest of the layers, achieving high resource efficiency to maintain consistent performance at bursts.

To fulfill Asymmetric Functions, we develop a serverless-oriented inference serving system called AsyFunc. Specifically, we make the following four contributions:

- We investigate the scaling issue of current serverless inference platforms caused by the time-consuming and resource-hungry model-loading process, and propose the key concept of *asymmetric functions* with different levels of model completeness (i.e., Body Function vs. Shadow Function) to solve this issue.
- We develop a heuristic algorithm for the model-level scaling (MLS) that adapts periodically and a priority-based heuristic algorithm for the layer-level scaling (LLS) that adapts on demand. Both of them aim to maximize the resource efficiency without hurting the performance. To make full use of the asymmetric functions, we devise an adaptive scheduling scheme to dispatch requests in real time.
- To enable fine-grained scaling at the layer level, we implement a high-performance and resource-efficient inference serving system AsyFunc[2] on top of Knative. For efficient coordination between Body and Shadow Functions during collaborative inference executions, we establish an efficient communication and synchronization mechanism that imposes negligible overhead on inference performance and resource consumption.
- We conduct extensive experiments to evaluate the performance of AsyFunc. Based on real-world traces, the evaluation results demonstrate that AsyFunc cuts down the memory resource consumption by up to 32.5% over an existing system and keeps the SLO violation rate at a low level despite the bursts.

2 BACKGROUND AND MOTIVATION

In this section, we first analyze the scaling issue of existing serverless inference platforms. Then, we illustrate the layer heterogeneity of DL models. Finally, we discuss the opportunities and challenges of layer-level scaling.

[2]Our project is open-sourced at https://github.com/peiqiangyu/AsyFunc.

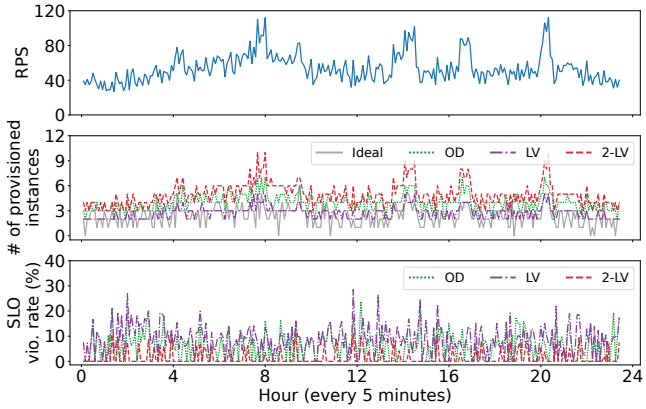


Figure 1: RPS over time, the number of provisioned instances, and SLO violation rate under different scaling policies. Existing policies involve inevitable trade-offs between resource efficiency and SLO satisfaction.

2.1 Scaling Issue of Serverless Inference

DL models consist of different types of neural network layers with different parameters (a.k.a. weights) [41]. The layers and their connections form a computationally directed acyclic graph (DAG), and inference executions are performed on these layers along the DAG. The DL models have recently achieved impressive performance in many areas, from image classification [21, 51] to natural language processing [13]. Major cloud providers, such as Amazon and Alibaba, have widely deployed DL models to provide inference services [42, 55]. However, online inference services are usually both latency-critical and resource-hungry, which leads to inevitable trade-offs between performance and resource efficiency, especially given the significant burstiness observed in the inference requests from users [7, 18, 27].

Serverless computing is considered to be a promising choice for handling bursty workloads due to its rapid elasticity and fine-grained billing [28, 34]. Therefore, model inference based on serverless platforms (i.e., serverless inference) has received widespread attention [2, 25, 60, 62]. In serverless inference, each DL model is deployed separately in a function instance (e.g., Docker [33], Firecracker [1]) that scales out/in as workload levels grow/drop. It is worth noting that before providing inference services, the function instance needs to be created first and then complete the model-loading process, i.e., loading the model file into the memory and initializing the model, which is very time-consuming. To maintain low inference latency in case of bursts, one common remedy is prewarming the instances and pre-loading the model beforehand [12, 16]. However, prewarming a sufficient number of instances for each DL model can cause a huge waste of resources, especially when there are many different models and under significant workload fluctuation. A recent study [35]

proposes a remedy that saves resources by only prewarming the instances without instantiating any model and thus can be shared by all models. In this way, the corresponding model will be loaded into the instance on demand. However, we notice that the time for loading the model is even longer than that for performing an inference execution (e.g., the gap is around 2× on CPU, which can be over 10× on GPU), which disables persistent real-time inference serving.

We attribute the above scaling issue to the time-consuming and resource-hungry behavior of the DL *model-loading process*. We conduct an experiment to verify this issue with a real-world trace from Twitter [3]. We select three classical scaling policies, namely on-demand scaling (OD), prediction-based scaling with the last value (LV) (i.e., the maximum resource demand during the last period) [9], and prediction-based scaling by multiplying the last value by a factor k (k-LV, k is 2 here), and also present the ideal scaling policy (Ideal) that provision functions on demand but assumes zero model-loading latency. The requests per second (RPS) during a day, the number of provisioned instances[3] and the SLO violation rate are plotted in Figure 1. We do not present the SLO violation rate of the Ideal policy since it is always zero.

Two conclusions can be summarized from this experiment: (1) The long model-loading process is likely to cause SLO violations if we scale out instances conservatively (the average SLO violation rates of the OD and LV are 6.0% and 8.7%, respectively); (2) The high demand for computing and storing will lead to a waste of resources if we scale out instances aggressively, where many instances are provisioned but actually unused (the average number of created instances of the k-LV is 5.0, about 2.1× of the actual demand as indicated by the Ideal policy). We notice that **the contradiction between SLO satisfaction and resource efficiency arises from the existing coarse-grained model-level scaling** that always loads the whole DL model into the new instances. Thus, we wonder whether we can reduce unnecessary resources without incurring perceptible SLO violations, by opening the black box of DL models and studying its internal layers.

2.2 Heterogeneous Behavior of the Layers

To improve the scaling efficiency, we identify the opportunity provided by the internal layers. It is worth noting that *batching* is a useful approach to increase the processing rate by grouping a number of user requests together, and the batch size denotes the number of grouped requests. A larger batch size can increase the processing rate but at the expense of latency performance. We measure the inference latency increase, the parameter size, and the loading time of each layer inside the EfficientNet-b5 model [52] when the batch size

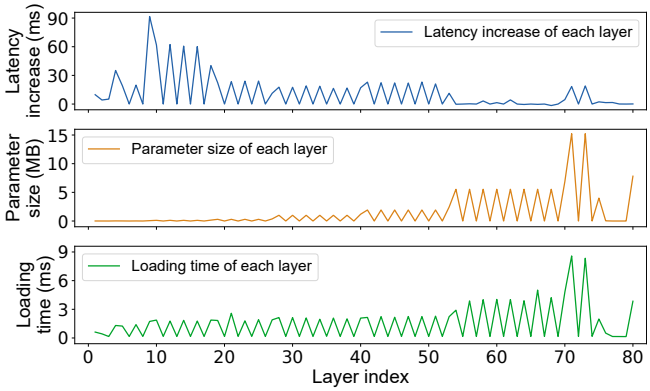


Figure 2: The latency increase of each layer in the EfficientNet-b5 model as the batch size grows. Surprisingly enough, the latency increase is almost negatively correlated with the parameter size and loading time.

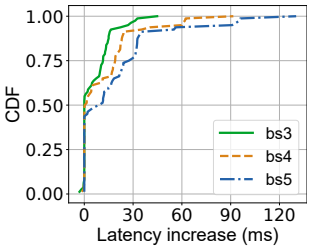


Figure 3: The CDF of latency increase of each layer when the batch size increases from 2 to 3, 4, and 5.

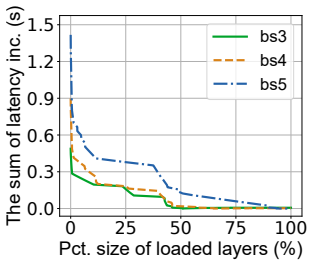


Figure 4: The increase in inference latency when loading different percentages of layers into the Shadow Function.

increases from 2 to 4, as shown in Figure 2. As we can see, the sensitivity of each layer to the computing resources varies. Specifically, **the inference latency increase is almost negatively correlated with the parameter size**, while **the model-loading latency is generally proportional to the parameter size**[4]. That is to say, both the memory usage and model-loading time can be reduced remarkably by loading a small number of resource-sensitive layers into the instance.

Figure 3 plots the cumulative distribution function (CDF) of the latency increase of each layer when the batch size increases. It is obvious that the latency increase of most layers is only marginal. For example, when the batch size doubles to four, although the largest latency increase is 92 ms, the 90th percentile of latency increase is around 24 ms, and the 50th percentile is nearly zero (note that the latency increase of the whole model is around 900 ms). For those non-resource-sensitive layers, increasing the batch size can increase the throughput largely but would not cause a significant latency

[3]We refer to function instances that pre-load DL models as *provisioned* instances.

[4]We provide more proof from other DL models and on other hardware of this anti-correlation phenomenon in Appendix A.

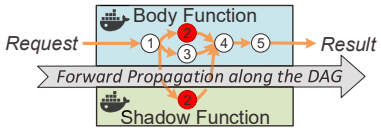


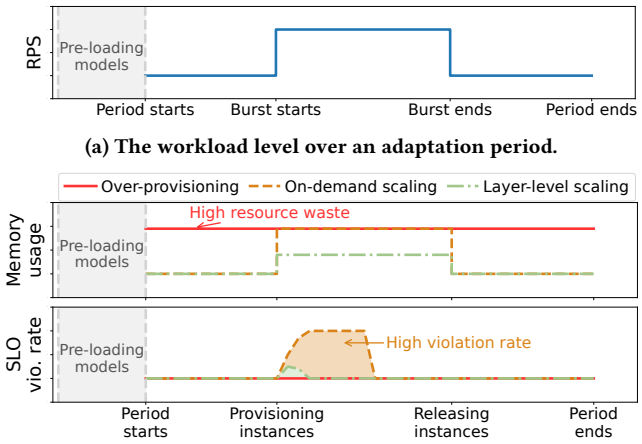
Figure 5: Illustration of collaboration between Body Function and Shadow Function in the layer-level scaling. The red circles denote the resource-sensitive layer, and computations on this layer are partially offloaded.

increase even if the amount of resources remains unchanged. Inspired by the above observation, we propose a fine-grained *layer-level scaling* mechanism to handle bursty workloads, where only some of the resource-sensitive layers will be loaded rather than a bulky complete model.

2.3 Opportunities of the Layer-level Scaling

Opportunities to handle bursts. In existing inference serving systems, to avoid perceptible latency increase because of a sudden spike of user requests, the scaling policy will re-direct the additional requests to new function instances that load a complete model. By contrast, when bursts arise, the layer-level scaling policy will allow new function instances to load only a small number of resource-sensitive layers in a timely manner which we call *Shadow Function*. As illustrated in Figure 5, during the inference process, the original function instances containing a complete model which we call *Body Function*, can offload part of the computations on those resource-sensitive layers to the Shadow Function. The two instances will perform computations on the common layers collaboratively and in parallel, and only the Body Function needs to perform computations on the remaining non-resource-sensitive layers. This helps satisfy sudden spikes in demand without heavy resource waste or performance degradation.

Evidence supporting the advantages. Figure 4 shows the latency increase when offloading computations on different numbers of layers to the Shadow Function as the number of requests (i.e., the batch size) grows from 2 to 3, 4, and 5. Note that the x-axis represents the ratio of the layers' parameter size to the whole model's parameter size, and the layers are sorted in descending order by their latency increase when chosen to load into the Shadow Function. Thus, "100%" indicates that the additional computations on the whole model are offloaded to the Shadow Function, which is equivalent to the model-level scaling policy. As observed, the latency increase grows slowly when computations on only a small number of layers get offloaded. In particular, the latency increase remains nearly zero when offloading computations on a few layers whose total parameter size is about half of the model's size. Also, such a small latency increase can be easily offset by allocating a few more cores to the Shadow Function temporarily.



(a) The workload level over an adaptation period.

(b) Trade-offs between resource consumption and SLO violation rate.

Figure 6: Brief comparison of three scaling policies to deal with bursts.

Preliminary demonstration. Figure 6 shows a comparison of the *layer-level scaling* policy with the model-level one (including *over-provisioning* and *on-demand scaling*) as workload level grows unexpectedly. The *over-provisioning* policy provisions enough instances with the whole model pre-loaded in case of sudden request surges, while the *on-demand scaling* policy provisions instances just for stable workloads (e.g., the most common workload level during a period) and provisions additional instances when bursts happen. By contrast, although the *layer-level scaling* policy also provisions instances only for stable workloads, it scales rather fast when bursts come by loading only a small number of resource-sensitive layers, achieving a good balance between SLO satisfaction and resource efficiency. According to our measurements, the model-loading time can be reduced by one to two orders of magnitude compared to that of loading a complete model.

Challenges to address. Despite proving to have great potential for serverless inference, the layer-level scaling mechanism faces critical challenges in coordinating Body and Shadow Functions. Firstly, it is necessary to adapt the functions' configuration (i.e., # of allocated CPU cores) and the set of layers loaded into the Shadow Function based on real-time workload levels, which affects both the inference latency and resource efficiency. Moreover, as offloading computations to another instance causes the inference workflow to span two instances, it consumes extra time to transfer the layers' output data, which varies from several KB to tens of MB. Thus, it is important to consider the output data size when deciding on the set of loaded layers. Finally, since current serverless platforms are a poor fit for supporting fine-grained coordination between functions, it is essential to devise an efficient communication and synchronization mechanism with minimum overhead and easy-to-use user interfaces.

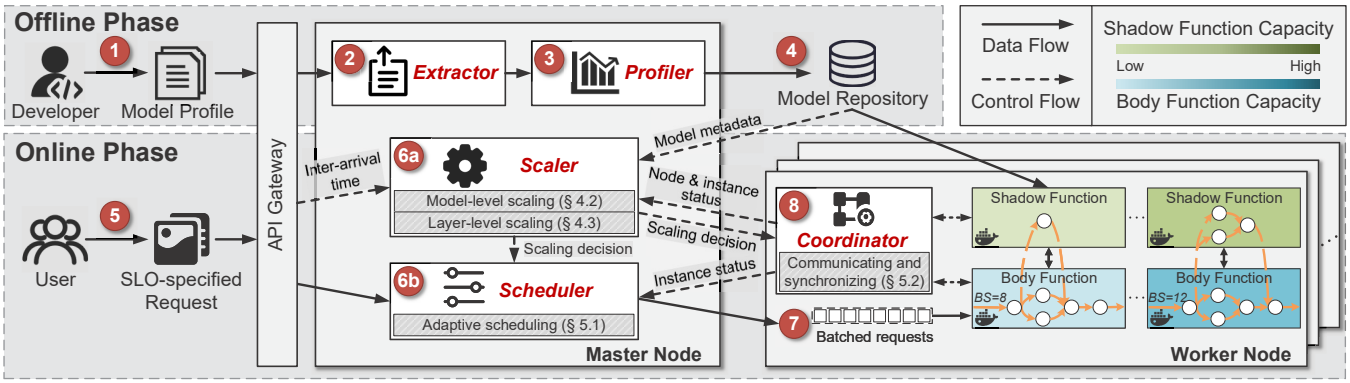


Figure 7: System overview of AsyFunc.

3 SYSTEM DESIGN

In this section, we present the design of AsyFunc, a serverless inference system that supports both coarse-grained model-level scaling and fine-grained layer-level scaling.

3.1 Design Philosophy

To take advantage of the fine-grained layer-level scaling discussed in Section 2.3, we develop a high-performance and resource-efficient serverless inference system with a new auto-scaling and scheduling engine. The main idea behind the engine lies in two aspects: (1) leveraging the *Body Function* that loads a complete model to handle stable inference workloads, and (2) leveraging the *Shadow Function* that loads a selective percentage of resource-sensitive layers to handle spiky inference workloads effortlessly. When bursts arise, the resource-sensitive parts of the DL model can be executed in a parallel manner using the Shadow Function, while other non-resource-sensitive parts are executed serially in the Body Function only. In this way, AsyFunc aims to achieve a good balance between SLO satisfaction and resource efficiency despite the ubiquitous bursts of inference requests.

3.2 System Overview

Based on the above design philosophy, we establish AsyFunc. Figure 7 shows the system overview of AsyFunc with the *Extractor*, *Profiler*, *Scaler*, *Scheduler*, and *Coordinator*.

In the offline phase, ❶ after the well-trained DL model is submitted to the platform, ❷ the *Extractor* automatically extracts its layer information (including the layer type and parameters) and structure information (i.e., the connections between adjacent layers). Then, ❸ the *Profiler* analyzes the resource sensitivity of each layer as well as other necessary metadata information. ❹ Finally, the layer, structure, and metadata information are stored in the model repository.

In the online phase, ❺ after receiving an SLO-specified request from users through the API Gateway, ❻a the *Scaler* on the master node collects the inter-arrival time information that will be used to calculate historical RPS for generating the scaling decision periodically. The scaling decision includes scaling out/in new Body Function instances periodically (i.e., the model-level scaling), and adjusting the maximum supported batch size of existing Body Function instances at bursts by scaling up/down Shadow Function instances (i.e., the layer-level scaling). Meanwhile, ❻b the *Scheduler* on the master node is responsible for dispatching the incoming request to an appropriate Body Function instance on worker nodes. ❼ Requests are first cached in the waiting queue to form a batch. Then, all requests in the queue are grouped together and sent to an instance on worker nodes. ❽ The Body Function instance can offload partial layers' computations to its paired Shadow Function instance at bursts, and the two instances are coordinated by the *Coordinator* on the worker node. In the following, we introduce each module in detail.

3.3 Extractor & Profiler

A DL model is composed of various types of layers that exhibit different behaviors. AsyFunc needs to extract layer information from model profiles submitted by developers and analyze the resource sensitivity of every layer. The *Extractor* and *Profiler* modules are responsible for these tasks.

The *Extractor* first parses the model profile and generates a profile for each layer as an individual file which contains the layer name, type, and parameters. In this way, layers can be selectively loaded as a sub-model into a Shadow Function instance at runtime, and the instance can perform inference computations only on these layers. Then, the *Extractor* reads the DAG structure information, i.e., connections between layers. With this connection information, the output data from the previous layer can be correctly passed to the next layer during an inference execution.

The *Profiler* first estimates the resource sensitivity of each layer based on its latency increase, groups adjacent layers with similar sensitivity into a *layer block*, and filters out layer blocks with low sensitivity or very large memory consumption. Instead of each individual layer, the layer block as a

whole will be selectively loaded into the Shadow Function to avoid frequent data transfer during an inference execution. For example, as shown in Figure 2, the layers numbered 8 to 15 will be merged into one block and the layer numbered 3 will be treated as another separate block. Next, the *Profiler* obtains the metadata information of each layer block by performing inference executions on these blocks separately, including the inference latency (l), parameter size (p), and output data size (d) under different numbers of CPU cores (c) and batch sizes (b), as a 5-tuple $< c, b, l, p, d >$.

3.4 Scaler

The *Scaler* is responsible for scaling out/in Body Functions periodically and scaling up/down Shadow Functions at bursts.

On the one hand, the *Scaler* estimates the average workload level during the next period based on historical inter-arrival time information. When it is expected to rise or decline, the *Scaler* will make scaling decisions through model-level scaling. Based on the model metadata and real-time node & instance status (including the number of free CPU cores on each node and the number of allocated CPU cores to each instance), the *Scaler* decides the configuration of the Body Function (i.e., # of CPU cores) and which worker node to accommodate it. On the other hand, when bursts arise unexpectedly within each period, the *Scaler* will make scaling decisions through layer-level scaling. The purpose of the layer-level scaling policy is to increase the maximum supported batch size of existing Body Function instances temporarily, by scaling up well-sized Shadow Function instances for a collaborative inference execution. Based on the model metadata and the reserved resources on each worker node, the *Scaler* decides which worker node to provision the Shadow Function and which layer blocks to load. The scaling details will be discussed in Section 4.

Note that the container pool is distributed among worker nodes. In other words, each worker node reserves one empty container that can load any layer blocks from any models on demand. In this way, the Shadow Function can be provisioned quickly by loading only the resource-sensitive layers without waiting for creating a container. Those reserved resources can also make room for other Body Functions when the remaining resources on that worker node are insufficient at high workload levels.

3.5 Scheduler & Coordinator

Both the *Scheduler* and *Coordinator* help realize real-time inference serving. First, the *Scheduler* is responsible for scheduling incoming requests to the best candidate instance, either the Body Function instance only or the Body and Shadow Function instance pair. Specifically, the *Scheduler* collects the instance status on all worker nodes, including the running state (i.e., idle or busy) and pairing state. As requests continue arriving in the waiting queue, the *Scheduler* forwards them as a batch to the best idle Body Function instance. We regard the instance that achieves the maximum throughput (i.e., the actual batch size divided by the inference latency) per CPU core as the best one. The scheduling details will be discussed in Section 5.1.

Second, the *Coordinator* is responsible for coordinating each instance pair during a collaborative inference execution in three ways: creating and destroying instances, facilitating efficient data transfer, and controlling correct synchronization. Note that the Body Function and its Shadow Function partner would be created on the same worker node as much as possible to avoid time-consuming cross-server communication. The details will be discussed in Section 5.2.

4 FINE-GRAINED SCALING MECHANISM

In this section, we describe in depth the design of model-level scaling and layer-level scaling in the *Scaler*.

4.1 Scaling Principle

First, we summarize the following two scaling principles:

(1) The model-level scaling policy for Body Functions aims at satisfying stable workloads which are represented by the *expected average RPS*.
(2) The layer-level scaling policy for Shadow Functions aims at satisfying spiking workloads which are represented by the *unexpected instantaneous RPS*.

Before making the model-level scaling decision periodically, the *Scaler* needs to evaluate whether the existing instances can meet the resource demand under the expected average RPS. Specifically, since it is Body Function instances that directly serve the user requests, the *Scaler* calculates the maximum supported RPS by all the Body Function instances paired with Shadow Function instances or not. Suppose there are n Body Function instances distributed in different worker nodes. For a Body Function instance i, we use b^i, t_q^i, and t_l^i to represent the batch size, queuing time, and the inference latency of the instance, respectively. As t_q^i approaches zero, the RPS that existing instances can handle (denoted as R_{max}) reaches the maximum. Thus, R_{max} can be calculated by:

$$R_{max} = \sum_{i=1}^{n} \max\{\frac{b^i}{t_l^i} \mid t_l^i \leq t_{SLO}\},\ \forall b^i \in \{1, 2, ..., B_{max}\}, \quad (1)$$

where t_{SLO} is the latency SLO. With Equation (1), the *Scaler* makes the following decisions:

(1) $R > \alpha R_{max}$. It means that existing instances cannot satisfy the predicted average RPS. The *Scaler* will scale out new Body Function instances by using the model-level scaling policy. We denote the residual RPS as R_k which is equal to $R - \alpha R_{max}$.

(2) $\beta R_{max} \leq R \leq \alpha R_{max}$. It indicates that existing instances can serve requests stably. The *Scaler* should take no action to avoid system instability caused by frequent scaling and node status switching.
(3) $R < \beta R_{max}$. It means that existing instances are beyond the demand under the predicted average RPS. The *Scaler* will release instances to save resources.

The *Scaler* makes the layer-level scaling decision each time a new request arrives. Once it detects that the instantaneous RPS grows largely, in other words, the waiting queue becomes crowded, it will provision Shadow Functions through the layer-level scaling policy. Since only the most resource-sensitive but memory-efficient layers will be loaded, the online model-loading process makes no perceptible impact on the latency performance. When the number of queuing requests drops back, the Shadow Functions will be released.

4.2 Model-Level Scaling

Based on Equation (1), the *Scaler* scales out Body Function instances to satisfy residual RPS or scales in to save resources.

However, it is nontrivial to decide on an appropriate configuration (i.e., # of CPU cores) as the RPS fluctuates. As the number of allocated CPU cores grows, the instance could achieve a lower inference latency or process a larger batch at a time with similar latency. In terms of resource efficiency, allocating more CPU cores would reduce the processing efficiency represented by the maximum throughput per core but increase the memory efficiency as fewer instances will be created. Thus, to select the best configuration, the model-level scaling policy is formulated as follows:

$$\min \sum_{i=1}^{N_1} \left(c^i + \rho m^i\right) \tag{2}$$

$$\text{s.t. } R_k \leq \alpha \sum_{i=1}^{N_1} \max\{\frac{b^i}{t_l^i} \mid t_l^i \leq t_{SLO}\},\ \forall b^i \in \{1, 2, ..., B_{max}\},$$

where c^i and m^i represent # of CPU cores and memory consumption of the instance $i \in 1 \ldots N_1$ to be created, and ρ is a normalizing factor related to the parameter size of the whole model. Considering that the Body Function instance loads a complete model in which m^i is a fixed value, the objective function (2) can be converted into $\min(\sum_i^{N_1} c^i + N_1 \cdot \rho \cdot m)$. To solve this problem, we develop a heuristic algorithm for model-level scaling (MLS) which decides the number of created instances N_1 and their configuration c^i.

As shown in Algorithm 1, when deciding to *scale out* (Line 2), for each core number and batch size (Lines 3-4), MLS first estimates the average longest service time that is equal to the average longest queuing time plus inference latency (Line 5). If it is within the latency SLO (Line 6), MLS calculates the **resource efficiency** that is defined as the maximum throughput per unit of computing resources plus that per unit of memory resources (Line 7) and obtains the maximum value for each core number (Lines 8-10). Finally, MLS chooses n configurations with the maximum resource efficiency (Line 11) and creates Body Function instances until $R \leq \alpha R_{max}$ (Lines 12-13). On the contrary, when deciding to *scale in* (Line 14), MLS will calculate the resource efficiency of each instance in X during the last period and sort them in ascending order (Line 15). Then, MLS will destroy them one by one until $R \geq \beta R_{max}$ (Lines 16-20).

Algorithm 1 Heuristic Algorithm for Model-Level Scaling

1: X: the set of existing Body Function instances;
   x_i: the i-th instance in X;
   $t_l^{c,b}$: the estimated inference latency when the number of CPU cores and batch size is c and b, respectively;
   $t_s^{c,b}$: the estimated longest service time when the number of CPU cores and batch size is c and b, respectively;
   c_{best}: the selected configurations of # of CPU cores of the newly created Body Function;
   η_{max}^c: the maximum achievable resource efficiency when the number of CPU cores is c, initialized as zero;
2: **if** $R > \alpha R_{max}$ **then**
3: **for** $c = 1, 2, \ldots, C_{max}$ **do**
4: **for** $b = B_{max}, B_{max} - 1, \ldots, 1$ **do**
5: $t_s^{c,b} = \frac{b-1}{R} + t_l^{c,b}$;
6: **if** $t_s^{c,b} \leq t_{SLO}$ **then**
7: $\eta = \frac{b}{c \cdot t_l^{c,b}} + \frac{b}{\rho m \cdot t_l^{c,b}}$;
8: **if** $\eta > \eta_{max}^c$ **then**
9: $\eta_{max}^c = \eta$;
10: **break**;
11: $c_{best} = \{c_1, c_2, \ldots, c_n \mid \eta_{max}^{c_1} \geq \eta_{max}^{c_2} \geq \cdots \geq \eta_{max}^{c_n} \geq \cdots \geq \eta_{max}^{c_{C_{max}}}\}$;
12: **while** $R > \alpha R_{max}$ **do**
13: Create a Body Function with $c \in c_{best}$ of cores on the worker node with the most available cores.
14: **else if** $R < \beta R_{max}$ **then**
15: Sort X by η in ascending order;
16: **for** $i = 1, 2, \ldots, |X|$ **do**
17: **if** $R < \beta R_{max}$ **then**
18: Destroy x_i;
19: **else**
20: **break**;

4.3 Layer-Level Scaling

As illustrated in Section 2, although the average RPS often varies slowly, the instantaneous RPS fluctuates severely and unexpectedly. When bursts arise, the Shadow Function helps improve R_{max} by increasing the maximum supported batch size of existing Body Function instances. In the following, we

Algorithm 2 Heuristic Algorithm for Layer-Level Scaling

1: Q: the waiting queue;
X: the set of unpaired Body Function instances sorted in descending order by the time they become idle;
x_i: the i-th instance in X;
c^i, m^i: # of CPU cores and the memory usage of the i-th instance in X;
c_a^i: the number of available CPU cores on the worker node where x_i resides;
b_B, b_S: the batch size supported by the Body and Shadow Functions, respectively, on the parallel part;
b: the batch size supported by the Body Function on the non-parallel part, where $b = b_B + b_S$;
L: the set of candidate layer blocks sorted in descending order by their resource sensitivity;
Φ: the set of selected layer blocks, where Φ is initialized as $\varnothing$ for each Shadow Function;
m_Φ: the memory consumption of the selected blocks Φ;
Φ_{best}: the best set of layer blocks for a Shadow Function;
η_{max}: the maximum achievable resource efficiency, initialized as zero;
2: **if** $|Q| \geq \gamma$ **then**
3: **for** $i = 1, 2, \ldots, |X|$ **do**
4: **if** $c_a^i \geq C_{max}/4$ **then**
5: $c = \min\{c_a^i, C_{max}/2\}$;
6: **for** ι in L **do**
7: $\phi \leftarrow \phi \cup \iota$;
8: Update m_Φ;
9: **for** $b = B_{max}, B_{max} - 1, \ldots, 1$ **do**
10: Select b_B, b_S to min $|t_{pB}^{c^i,b_B} - t_{pS}^{c,b-b_B}|$;
11: $t_l^{c,b} = \max\{t_{np}^{c^i,b} + t_{pB}^{c^i,b_B}, t_{np}^{c^i,b} + t_{pS}^{c,b_S}\}$;
12: $t_s^{c,b} = t_{load}(\phi) + t_l^{c,b}$;
13: **if** $t_s^{c,b} \leq t_{SLO}$ **then**
14: $\eta = \frac{b}{(c^i+c)\cdot t_l^{c,b}} + \frac{b}{\rho(m^i+m_\Phi)\cdot t_l^{c,b}}$;
15: **if** $\eta > \eta_{max}$ **then**
16: $\eta_{max} = \eta$;
17: $\phi_{best} = \phi$;
18: **break**;
19: Provision a Shadow Function instance that loads the layer blocks of ϕ_{best} on the worker node.

propose a layer-level scaling policy that scales up Shadow Function instances in a timely manner to satisfy sudden demands with minimum resource consumption.

Each Body Function instance can be paired with one Shadow Function instance. Suppose N_2 represents the number of unpaired Body Function instances on all worker nodes, so there are at most N_2 Shadow Function instances to be scaled up, where the *Scaler* needs to determine their configurations, including # of CPU cores and the set of loaded layer blocks. The layer-level scaling policy is formulated as follows:

$$\min \sum_{j=1}^{N_2} \left(c^j + \rho m^j\right)$$

$$\text{s.t. } R_{burst} \leq \sum_{j=1}^{N_2} \max\{\frac{b^j}{t_l^j} \mid t_l^j \leq t_{SLO}\},\ \forall b^j \in \{1, 2, ..., B_{max}\},$$

$$t_l^j = \max\{t_{np}^j + t_{pB}^j, t_{np}^j + t_{pS}^j\},\ \forall j \in \{1, 2, ..., N_2\}, \quad (3)$$

where c^j and m^j represent # of CPU cores and memory consumption of the instance $j \in 1 \ldots N_2$, and t_{np}^j, t_{pB}^j, and t_{pS}^j refer to the inference latency on the non-parallel part of the Body Function, the inference latency on the parallel part of the Body Function, and the inference latency on the parallel part of the Shadow Function plus the data transmission time. Taking the case shown in Figure 5 as an example, t_{np}^j refers to the inference latency of layers numbered 1, 4, and 5, t_{pB}^j is the inference latency of layers numbered 2 and 3 in the Body Function instance, and t_{pS}^j is the inference latency of the layer numbered 2 in the Shadow Function instance as well as the data transmission time. The overall inference latency is the highest one of $(t_{np}^j + t_{pB}^j)$ and $(t_{np}^j + t_{pS}^j)$.

As an NP-hard problem, we convert it into a heuristic algorithm for realizing the layer-level scaling (LLS). First, LLS gives priority to Body Functions with earlier time to become idle to be paired with Shadow Functions so that the Shadow Functions can be fully used soon after. Secondly, layer blocks with higher resource sensitivity will be selected first. Thirdly, the number of CPU cores allocated to Shadow Functions is fixed; however, if the available CPU cores are insufficient on the node where the Body Function resides, all the remaining cores will be allocated. As shown in Algorithm 2, for each Body Function instance (Line 2), LLS first judges whether it is necessary to scale up according to the length of Q (Line 2). If so, for each Body Function, LLS determines the number of cores for the Shadow Function, adds one layer block each time, and updates the memory usage (Lines 3-8). Next, for each batch size, LLS determines b_B and b_S to guarantee the two parallel parts almost finish at the same time (Lines 9-10). Then, LLS estimates the inference latency based on Equation (3) and the service time that is equal to the model-loading time t_{load} and inference latency (Lines 11-12). If the latency SLO can be satisfied (Line 13), LLS chooses the configuration that maximizes the resource efficiency (Lines 14-18) to provision a Shadow Function instance (Line 19). For the next Body Function, LLS repeats the above steps until the number of newly provisioned Shadow Functions exceeds a pre-defined value κ. Similar to MLS, as the burst passes off, LLS scales down by destroying Shadow Function one by one whose η is the smallest.

Algorithm 3 Adaptive Scheduling Algorithm

1: Q: the waiting queue;
   X: the set of idle Body Function instances sorted in ascending order by # of allocated CPU cores;
   x_i: the i-th instance in X;
   t_w: the longest waiting time of queuing requests in Q;
   t_l^i, t_s^i: the estimated inference latency and the estimated longest service time of queuing requests if choosing x_i;
2: Flag = FALSE;
3: **while** a request arrives in Q **do**
4: **for** $i = 0, 1, \ldots, |X|$ **do**
5: $t_s^i = t_w + t_l^i$;
6: Record t_s^i in T;
7: **if** $t_s^i \leq t_{SLO}$ **then**
8: Schedule Q to x_i and clear Q;
9: Flag = TRUE;
10: **break**;
11: **if** not Flag **then**
12: **if** $t_s^i > t_{SLO}, \forall t_s^i \in T$, **then**
13: Choose an instance x_i with the largest i;
14: Schedule a portion of the latest requests in Q to x_i and clear Q;
15: **else**
16: Wait for the next request;

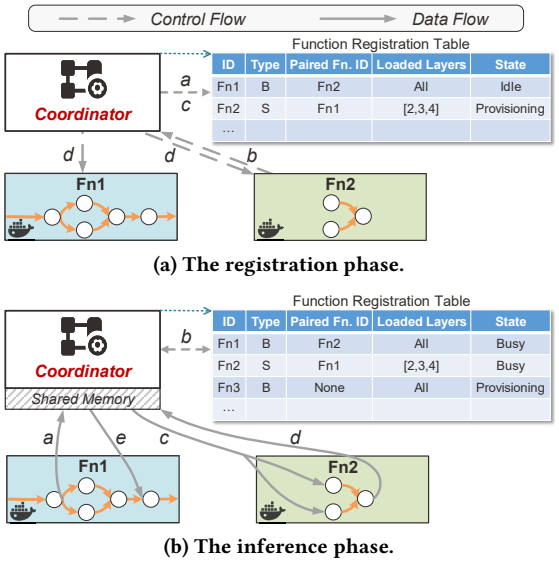
5 REAL-TIME INFERENCE SERVING

In this section, we present the scheduling algorithm to dispatch incoming requests to the best instance and the coordination mechanism to manage Body and Shadow Function instances, both of which ensure real-time inference serving.

5.1 Adaptive Scheduling

As requests arrive in the waiting queue, the *Scheduler* needs to decide on both *the appropriate time* to dispatch all these requests as a batch and *the best instance* to serve them. The former impacts both the waiting time and inference time as the number of requests in the waiting queue increases, while the latter only impacts the inference time. To maximize the **processing efficiency** (defined as the actual throughput per core) while keeping a low SLO violation rate, the *Scheduler* makes decisions based on the following observation: Smaller instances generate higher processing efficiency, while larger instances are more robust to workload fluctuations as they can process a larger batch size a time.

Hence, we develop an adaptive scheduling algorithm that prioritizes smaller instances and leaves behind larger ones for dealing with bursts. As shown in Algorithm 3, as a new request arrives (Line 3), the *Scheduler* first calculates the estimated longest service time if choosing the instance x_i (Lines 4-5), and records it in a list (Line 6). If that instance satisfies the SLO, the *Scheduler* will dispatch all the requests in Q at once and change the flag (Lines 7-10). If the requests are not successfully scheduled in the end (Line 11), there are two cases: (1) None of the idle instances can satisfy the SLO. Hence, to ensure the SLO of as many as requests, the *Scheduler* will selectively dispatch a portion of the latest requests in the queue to an idle instance with the highest configuration (i.e., the largest i value) and drop other earlier requests [27] (Lines 12-14); (2) There is no idle instance at the moment. Therefore, the *Scheduler* will wait for the next request (Lines 15-16).



(a) The registration phase.

(b) The inference phase.

Figure 8: Demonstration of the coordination process.

5.2 Coordination Mechanism

As introduced in Section 3.5, the *Coordinator* is distributed on each worker node and manages the life cycle of all functions. Specifically, AsyFunc maintains a Function Registration Table (FRT) that records the metadata information of each function, including the function ID, function type, ID of its paired function, indexes of loaded DL model layers, and function status. Upon receiving a scaling decision from the *Scaler*, the *Coordinator* will create or destroy corresponding functions and update the node and instance status as well as the FRT. At runtime, AsyFunc uses shared memory for fast data transmission between Body and Shadow Functions.

As plotted in Figure 8a, when receiving a layer-level scaling decision, ⓐ the *Coordinator* will parse the received message, insert a record into the FRT, and provision a Shadow Function. ⓑ When the provisioning process completes, ⓒ the *Coordinator* updates the function status and ⓓ sends a registration success message to the corresponding paired functions so that they start to process inference requests collaboratively. As plotted in Figure 8b, when the Body Function

receives an invocation and then finishes the computations on the non-parallel part, ⓐ it sends an offloading message to the *Coordinator* with the intermediate data to be processed in parallel. Meanwhile, the *Coordinator* ⓑ queries the registration table, obtains the ID of the paired Shadow Function, and ⓒ then forwards the intermediate data to the Shadow Function. Once the processing is complete, ⓓ the Shadow Function sends a completion message to the *Coordinator* with the result data. ⓔ After obtaining the result data from both the *Coordinator* and local processing, the Body Function merges them and continues to perform further computations.

6 IMPLEMENTATION

We implement a real system prototype of AsyFunc on top of an open-source serverless platform based on Knative [22] with about 3k lines of code in Python and C++. First, to obtain model metadata from benchmark models (as listed later in Table 2), we implement the *Extractor* and *Profiler* as a Python toolkit. Second, we implement the *API Gateway*, *Scaler*, *Scheduler*, and *Coordinator* to deploy and invoke functions through Knative Serving Service and Knative Serving Ingress. These modules realize the following functions: collecting and preprocessing user requests, and making scaling and scheduling decisions. We detail the implementation of the *Coordinator* as follows.

The *Coordinator* creates a TCP service to send/receive notifications to/from each function and leverages the memory filesystem [50] to transfer intermediate data during an inference execution. The shared memory is viewed as a directory and is mapped to the memory area of each function instance on the same node using the Linux kernel's `mmap` system call [38], where the cross-function communication is established through reading and writing files in this directory. At runtime, the *Coordinator* monitors file system events in the shared memory using the inotify [31] API in the Linux kernel. The file system events include creating or destroying functions and transferring data between functions. Specifically, (1) Once a new function is provisioned, it records its metadata information as a new file in the memory. When the *Coordinator* identifies this event, it updates the record in the FRT with the metadata and then deletes the file. (2) When a Body Function needs to call a Shadow Function for offloading computations, it writes the offloaded data as a new file in the memory, which is differentiated by the instance's ID. Then, the *Coordinator* uses the ID to find the corresponding ID of the paired Shadow Function in the FRT and instructs the Shadow Function to read the data and then delete the file. At last, the Shadow Function returns the inference result to the Body Function in the same way. According to our experiments, it takes about 0.32 ms to write and read 1 MB of the data through the shared memory.

For application developers, they only need to make three changes to their code for using AsyFunc, including (1) import AsyFunc's Python package, (2) use `getModel(model)` for extracting and profiling of the DL model, and (3) use `inference(inputs,SLO)` for executing the inference.

7 EVALUATION

In this section, we evaluate the performance of AsyFunc through extensive experiments. We first introduce the experimental setup and then show the experimental results compared to the state of the art.

7.1 Experimental Setup

Environment configuration. We deploy AsyFunc in a local private cluster to provide inference services, and the specification of each node can be found in Table 1. To accelerate the experiments on a 30-node cluster, we switch our implementation to the simulation mode as previous work does [27], where the inference latency under different configurations and the data transmission time under different data sizes are measured on the physical node.

Table 1: Experimental environment.

Item	Specification
CPU device	Intel Xeon E5-2697
Number of cores	36
Base frequency	2.30 GHz
Memory capacity	64 GB
Operating system	Ubuntu 18.04
DL framework	PyTorch 1.12

System parameters. The maximum number of allocated CPU cores C_{max} and the maximum supported batch size B_{max} of a Body Function are set to 16 and 4, respectively, because it could be ineffective to further reduce latency by allocating more cores. Further, the scaling parameters α and β are set to 0.8 and 0.6, respectively, and the adaptation period of the model-level scaling is 60 s. The layer-level scaling thresholds γ and κ are set to 2 and 10, respectively.

User requests. We use real-world traces from Twitter [3] to generate user requests. It represents a typical arrival of tweets for sentiment analysis, which has been widely used for evaluating inference systems [2, 62]. Figure 1 shows the RPS during a day which exhibits obvious burst characteristics.

Inference workloads. We select six representative DL models from different families as the inference workloads, including InceptionV3, ResNet50, EfficientNet-b5, SSD300, YOLOv8x, and VGG16, which are built on PyTorch [39]. We set their latency SLOs based on the inference latency when the number of allocated cores is C_{max} and the batch size is B_{max}, for the following experiments. Their values and the model details are shown in Table 2.

Table 2: Benchmark DL models.

Model	Parameter size	# of layers	# of layer blocks	SLO (s)
InceptionV3 [51]	91 MB	104	5	0.3
ResNet50 [21]	98 MB	126	15	0.5
EfficientNet-b5 [52]	117 MB	80	7	2.0
SSD300 [29]	136 MB	65	6	1.2
YOLOv8x [43]	261 MB	119	12	4.0
VGG16 [49]	528 MB	39	6	0.8

Performance metrics and baseline systems. We compare AsyFunc with a state-of-the-art serverless inference system BATCH [2] under two metrics, including resource efficiency and the SLO violation rate. For fairness of comparison, we realize BATCH's scaling policy on top of AsyFunc's implementation, which determines the functions' configuration offline and scales at a coarse-grained model level. To show the benefit of the layer-level scaling, we disable the layer-level scaling ability and denote this system as AsyFunc$^-$.

7.2 Overall Performance

Computing resource efficiency. We first calculate the computing resource usage of all provisioned function instances in the cluster in each adaptation period. Figure 9 shows the violin plot of the CPU usage by six DL models under AsyFunc, AsyFunc$^-$, and BATCH, where the text labels show the average values. As we can see, AsyFunc$^-$ reduces the CPU usage by 7.0%~20.9% as compared with BATCH due to its adaptive scaling and scheduling ability, which adjusts the configuration of functions and the batch size flexibly to make full use of CPU resources. On the contrary, BATCH adjusts the configuration and the batch size after a long period and restricts the configuration to just one value in each period, making it hard to capture short-term workload fluctuations.

AsyFunc further lowers the CPU usage by 0.7%~13.0% than AsyFunc$^-$ and 9.6%~31.1% (19.0% on average) than BATCH thanks to its layer-level scaling ability. Given the high model-loading time, BATCH tends to provision sufficient functions in advance for maintaining SLOs. By comparison, through scaling at a fine-grained layer level, the model-loading overhead can be largely reduced so that the Shadow Function can be provisioned in a timely manner in case of possible performance degradation as bursts come unexpectedly. For the InceptionV3 and ResNet50 models, the additional savings brought by the layer-level scaling seem marginal. This is because they are very lightweight (see their SLOs) and consume fewer CPU resources to finish inference tasks than other models. Thus, provisioning Shadow Functions for them may bring marginal performance gains. Nevertheless, the layer-level scaling is able to cut down resource usage considerably as the model grows large in terms

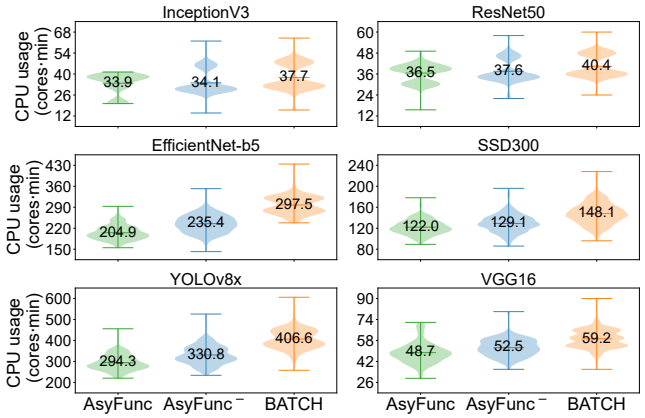


Figure 9: The violin plot of the CPU usage.

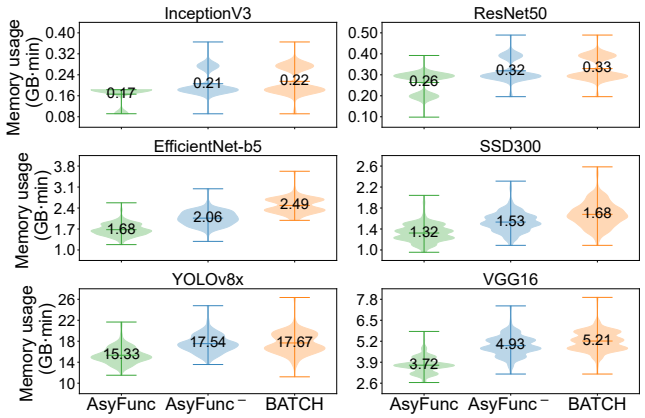


Figure 10: The violin plot of the memory usage.

of computational complexity, such as EfficientNet-b5 and YOLOv8x, whose additional resource savings are about 12%.

Memory resource efficiency. As discussed in Section 2.2, the proposed layer-level scaling is especially friendly to memory resources. Next, we calculate the memory resource usage in each period. As plotted in Figure 10, AsyFunc$^-$ lowers the memory footprint by 0.7%~17.0% than BATCH, while AsyFunc furthers reduces it by 12.6%~24.6% than AsyFunc$^-$ and 13.3%~32.5% (23.1% on average) than BATCH. Such significant resource savings arise from AsyFunc's layer-level scaling ability that takes advantage of the anti-correlation phenomenon as presented in Section 2.2. To maintain a low SLO violation rate in the face of burstiness, the Shadow Functions only load a portion of resource-sensitive layers adaptively that consume only a few memory resources. This is especially promising for future AI accelerators whose on-chip memory is usually scarce as compared to the host memory.

SLO violation rate. Finally, we calculate the SLO violation rate by counting the percentage of requests whose SLO constraint is not satisfied. Figure 11 shows the violin plot of the SLO violation rate of the six DL models. As compared to BATCH, there is an insignificant or no increase

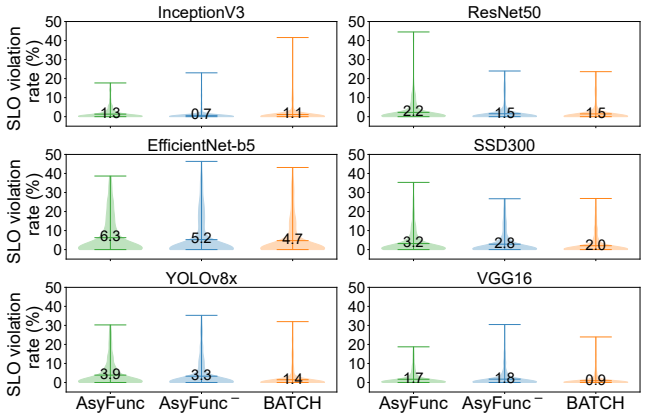


Figure 11: The violin plot of the SLO violation rate.

in the violation rate under AsyFunc and AsyFunc$^-$ thanks to the lightweight layer-level scaling that can scale up a Shadow Function instance within a few milliseconds. In this way, the Body Function instance can offload additional computations to the Shadow Function instance and thus serve more requests when the workload level suddenly grows high. In summary, to achieve similarly low SLO violation rates, the fine-grained scaling and adaptive scheduling capabilities of AsyFunc can save abundant computing and memory resources, which potentially helps improve the function density and thus save costs for datacenter facilities.

7.3 Extension of AsyFunc

Serving transformer-based models. According to our investigation, there may be no significant anti-correlation phenomenon in transformer-based models. Nevertheless, AsyFunc still outperforms the state of the art thanks to its layer-level scaling support that achieves an adaptive trade-off between performance and resource efficiency. According to our experiments on the Vision Transformer (ViT) model [15], AsyFunc consumes 6.6% fewer memory resources than BATCH while keeping a low SLO violation rate.

Supporting GPU hardware. Nowadays, it is very common to use heterogeneous hardware, such as GPUs, for inference executions. As major serverless platforms [5, 17] do not support heterogeneous hardware at the moment, we do not include advanced heterogeneous hardware in our current version. Nevertheless, it complements AsyFunc's fine-grained scaling capabilities at the layer level. Here, we briefly illustrate how AsyFunc can be equipped with GPU support. First, in the scaling algorithm, the GPU resources can be represented as the number of SMs (vs. # of CPU cores) and/or the GPU memory usage (vs. the host memory) with virtualization technologies like virtual GPU and Multi-Instance GPU. Second, for the implementation, the CPU functions can be replaced by GPU-supported ones, such as *nvidia-docker*. There are indeed some technical issues to overcome, such as efficient coordination between CPU functions and GPU functions and the logic of data exchange between the Body Function and Shadow Function. It would be an interesting future work to extend our approach to these scenarios in practice. In the Appendix, we show that on GPU, the overhead of loading models becomes more severe, where the inference latency is much lower than on CPU, but the model-loading time is higher. Thus, we expect the revealed anti-correlation phenomenon may benefit the GPU scenarios more.

8 RELATED WORK

Conventional model serving. Many efforts have been devoted to designing efficient scheduling mechanisms for model serving to achieve various objectives, e.g., low end-to-end latency [11, 19, 27, 58], high throughput [14, 20, 27], high resource efficiency [30, 56], and good fairness [24]. However, they only focus on the application level, without digging into the bottom inference platforms, leaving a large performance gap to be filled. Additionally, facing the widespread burstiness in the production environment, most of these prior arts cannot adapt to the fluctuating workload efficiently, leading to a significant trade-off between performance and resource efficiency. Although AlpaServe [27] employs statistical multiplexing with model parallelism to reduce serving latency for bursty workloads, the biggest difference is that it focuses on static provisioning through automatic parallelization and placement of models, ignoring the auto-scaling capability that the serverless native provides.

Serverless inference. Driven by the development of serverless computing, many works [2, 7, 25, 60–62] have attempted to deploy efficient machine learning inference serving on serverless platforms to take advantage of its rapid elasticity and fine-grained billing ability. Nevertheless, although these techniques consider an auto-scaling setting, most of them regard the DL model as a complete black box [2, 7, 60, 62], which leads to resource inefficiency at a coarse-grained model-level scaling when directly applied for serverless inference serving. Recent literature has further attempted to open the black box of DL models to reduce resource footprint through tensor sharing [25], but it still lacks generalizability in dealing with various model families as it relies on the layer sameness that mainly exists among model variants in the same family. By contrast, AsyFunc exploits the differences between any model layers. Gillis [61] is another work to open the box, which focuses on partitioning large DL models so that they can fit into small functions. By comparison, AsyFunc focuses on the scaling problem of serverless platforms and only scales out resource-sensitive layers to perform additional computations as bursts arise.

Serverless cold start. The cold start issue of serverless functions has been an active research topic in recent years [1, 6, 26, 37, 46–48, 53, 54]. These works typically adopt two different technical paths, including (1) avoiding cold starts based on predictive prewarming techniques [6, 46] and container keep-alive strategies [37, 47], and (2) reducing the latency of a single cold start based on snapshots [53, 54] and lightweight runtime techniques [1, 26, 48]. Taking industrial practices as an example, Azure deploys a practical hybrid histogram policy to dynamically decide the prewarming and keep-alive windows by characterizing the serverless workloads, which significantly reduces the number of cold starts but consumes fewer resources [47]. However, these works focus on platform-level cold starts, while AsyFunc deals with "cold starts" specifically for inference applications. They are complementary to AsyFunc to further improve the resource efficiency of real-time inference serving for cloud providers.

9 CONCLUSION

In this paper, we propose a high-performance and resource-efficient serverless inference system called AsyFunc for handling bursty DL workloads efficiently. By analyzing the impact of DL models' completeness, we find that the layer's sensitivity to computational resources is largely anti-correlated with its parameter size, and the latter further determines the memory usage and model-loading time. Driven by this phenomenon, we propose a new concept of *asymmetric functions* where the original Body Function still loads a complete model to satisfy stable demands, while the proposed lightweight Shadow Function loads only a portion of resource-sensitive layers to handle surging demands. On top of Knative, AsyFunc is equipped with layer-level scaling capability to achieve the above goal. The evaluation results show that AsyFunc outperforms the state-of-the-art system by up to 32.5% in memory resource efficiency while meeting the SLO target despite workload burstiness.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Ingo Müller for their valuable feedback. This work was funded in part by National Key Research & Development (R&D) Plan under grant 2022YFB4501703, in part by The Major Key Project of PCL (PCL2022A05), and in part by National Science Foundation of China under grant 62232012.

A MORE PROOF OF THE ANTI-CORRELATION PHENOMENON

To show the widespread anti-correlation phenomenon of DL models, we provide more proof from other model families and on other hardware here.

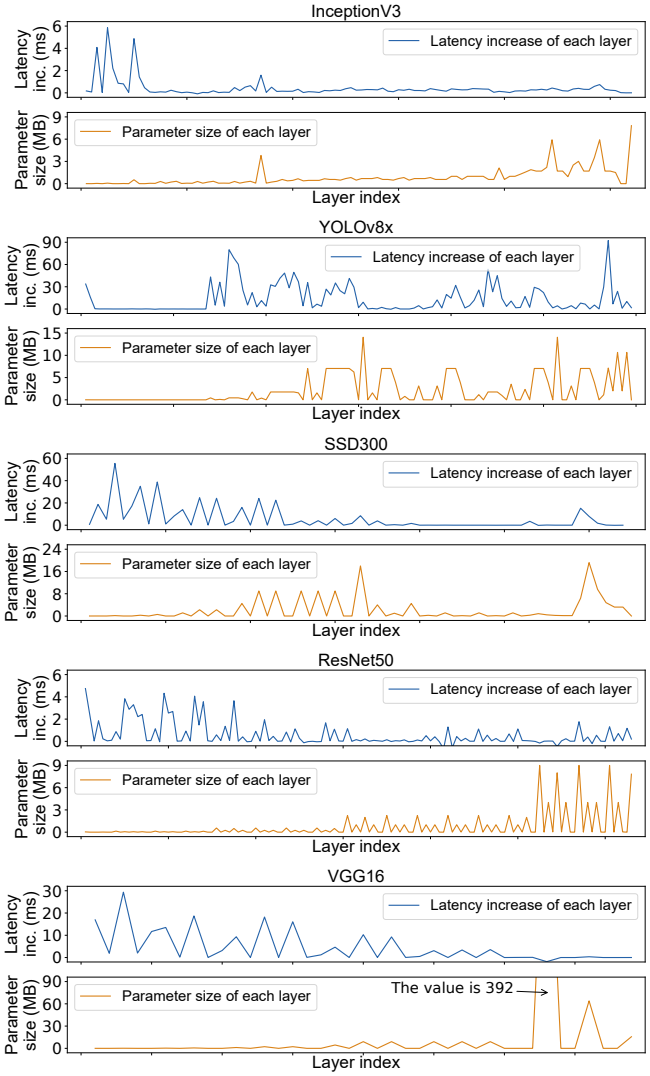


Figure 12: The anti-correlation phenomenon of other models (core number = 12, bs = 2 → 4).

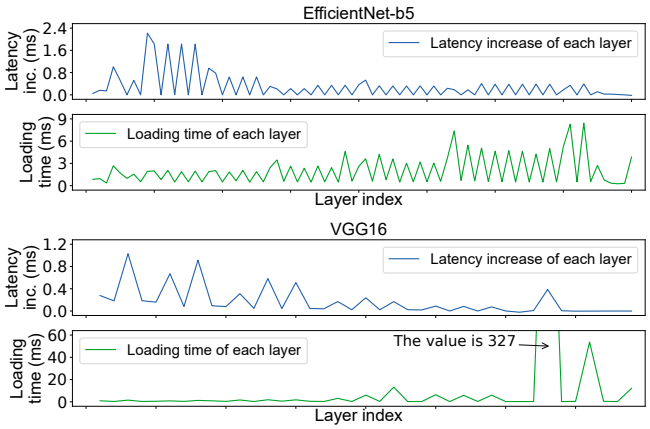


Figure 13: The anti-correlation phenomenon on an NVIDIA Quadro RTX 6000 GPU (bs = 6 → 10).

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. 419–434.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[3] Archive Team. [n. d.]. Archive Team: The Twitter Stream Grab. https://archive.org/details/twitterstream[Online Accessed, 28-Sept-2023].

[4] AWS. [n. d.]. Alexa skills. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html[Online Accessed, 28-Sept-2023].

[5] AWS. [n. d.]. Amazon SageMaker. https://aws.amazon.com/sagemaker/[Online Accessed, 28-Sept-2023].

[6] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 153–167.

[7] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. BARISTA: Efficient and scalable serverless serving system for deep learning prediction services. In *Proceedings of the IEEE International Conference on Cloud Engineering*. IEEE, 23–33.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[9] Brad Calder, Glenn Reinman, and Dean M Tullsen. 1999. Selective value prediction. In *Proceedings of the 26th annual international symposium on computer architecture*. 64–74.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. 613–627.

[11] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference*. 183–198.

[12] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[16] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[17] Anirudh Garg. [n. d.]. Why use Azure Functions for ML inference? https://techcommunity.microsoft.com/t5/apps-on-azure-blog/why-use-azure-functions-for-ml-inference/ba-p/1416728.

[18] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*. 109–120.

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 443–462.

[20] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture*. IEEE, 27–40.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[22] Knative. [n. d.]. Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/docs/[Online Accessed, 28-Sept-2023].

[23] Oleksiy Kovyrin. [n. d.]. Make Data Useful by Greg Linden. https://www.scribd.com/doc/4970486/[Online Accessed, 28-Sept-2023].

[24] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[25] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference*.

[26] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference*. 53–68.

[27] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*. 663–679.

[28] Yanying Lin, Kejiang Ye, Yongkang Li, Peng Lin, Yingfei Tang, and Chengzhong Xu. 2021. BBServerless: A Bursty Traffic Benchmark for Serverless. In *Proceedings of the International Conference on Cloud Computing*. Springer, 45–60.

[29] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *Proceedings of the European conference on computer vision*. Springer, 21–37.

[30] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant

deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.

[31] Robert Love. 2005. Kernel korner: Intro to inotify. *Linux Journal* 2005, 139 (2005), 8.

[32] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. 2020. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro* 40, 2 (2020), 8–16.

[33] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014).

[34] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A Chien. 2019. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*. 1–6.

[35] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference*. 57–70.

[36] OpenAI. [n. d.]. Introducing ChatGPT. https://openai.com/blog/chatgpt[Online Accessed, 28-Sept-2023].

[37] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. 2022. Retention-aware container caching for serverless edge computing. *Proceedings of the IEEE International Conference on Computer Communications* (2022).

[38] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 USENIX Annual Technical Conference*. 813–827.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32.

[40] Qiangyu Pei, Shutong Chen, Qixia Zhang, Xinhui Zhu, Fangming Liu, Ziyang Jia, Yishuo Wang, and Yongjie Yuan. 2022. CoolEdge: hotspot-relievable warm water cooling for energy-efficient edge datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 814–829.

[41] Pytorch. [n. d.]. Saving and Loading Models. https://pytorch.org/tutorials/beginner/saving_loading_models.html[Online Accessed, 28-Sept-2023].

[42] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*.

[43] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. 2023. Real-Time Flying Object Detection with YOLOv8. *arXiv preprint arXiv:2305.09972* (2023).

[44] Mariliis Retter. [n. d.]. Serverless Case Study – Netflix. https://dashbird.io/blog/serverless-case-study-netflix/[Online Accessed, 28-Sept-2023].

[45] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference*. 397–411.

[46] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.

[47] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference*. 205–218.

[48] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference*. 419–433.

[49] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[50] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *Proceedings of the autumn 1990 EUUG Conference*. 241–248.

[51] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. [n. d.]. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, title=Rethinking the Inception Architecture for Computer Vision, year=2016, pages=2818-2826*.

[52] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the International Conference on Machine Learning*. PMLR, 6105–6114.

[53] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[54] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference*. 1–16.

[55] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. 945–960.

[56] Jing Wu, Lin Wang, Qiangyu Pei, Xingqi Cui, Fangming Liu, and Tingting Yang. 2022. HiTDL: High-throughput deep learning inference at the hybrid mobile edge. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4499–4514.

[57] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2021. Serverless Data Science-Are We There Yet? A Case Study of Model Serving. *arXiv e-prints* (2021), arXiv–2103.

[58] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE, 392–405.

[59] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. 2021. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference*. 37–53.

[60] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 768–781.

[61] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proceedings of the IEEE 41st International Conference on Distributed Computing Systems*. IEEE, 138–148.

[62] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine

Learning Inference Serving. In *2019 USENIX Annual Technical Conference*. 1049–1062.